How to Write a plugin for JMeter

Authors

Mike Stover, Peter Lines

Revision 1.0 November 2004

Table of Contents

How to write a plugin for JMeter

Introduction from Peter Lin

On more than one occasion, users have complained JMeter's developer documentation is out of date and not very useful. In an effort to make it easier for developers, I decided to write a simple step-by-step tutorial. When I mentioned this to mike, he had some ideas about what the tutorial should cover.

Before we dive into the tutorial, I'd like to say writing a plugin isn't necessarily easy, even for someone with several years of java experience. The first extension I wrote for JMeter was a simple utility to parse HTTP access logs and produce requests in XML format. It wasn't really a plugin, since it was a stand alone command line utility. My first real plugin for JMeter was the webservice sampler. I was working on a .NET project and needed to stress test a webservice. Most of the commercial tools out there for testing .NET webservices suck and cost too much. Rather than fork over several hundred dollars for a lame testing tool, or a couple thousand dollars for a good one, I decided it was easier and cheaper to write a plugin for JMeter.

After a two weeks of coding on my free time, I had a working prototype using Apache Soap driver. I submitted it back to JMeter and mike asked me if I want to be a committer. I had contributed patches to Jakarta JSTL and tomcat in the past, so I considered it an honor. Since then, I've written the access log sampler, Tomcat 5 monitor and distribution graph. Mike has since then improved the access log sampler tremendously and made it much more useful.

Introduction from Mike Stover

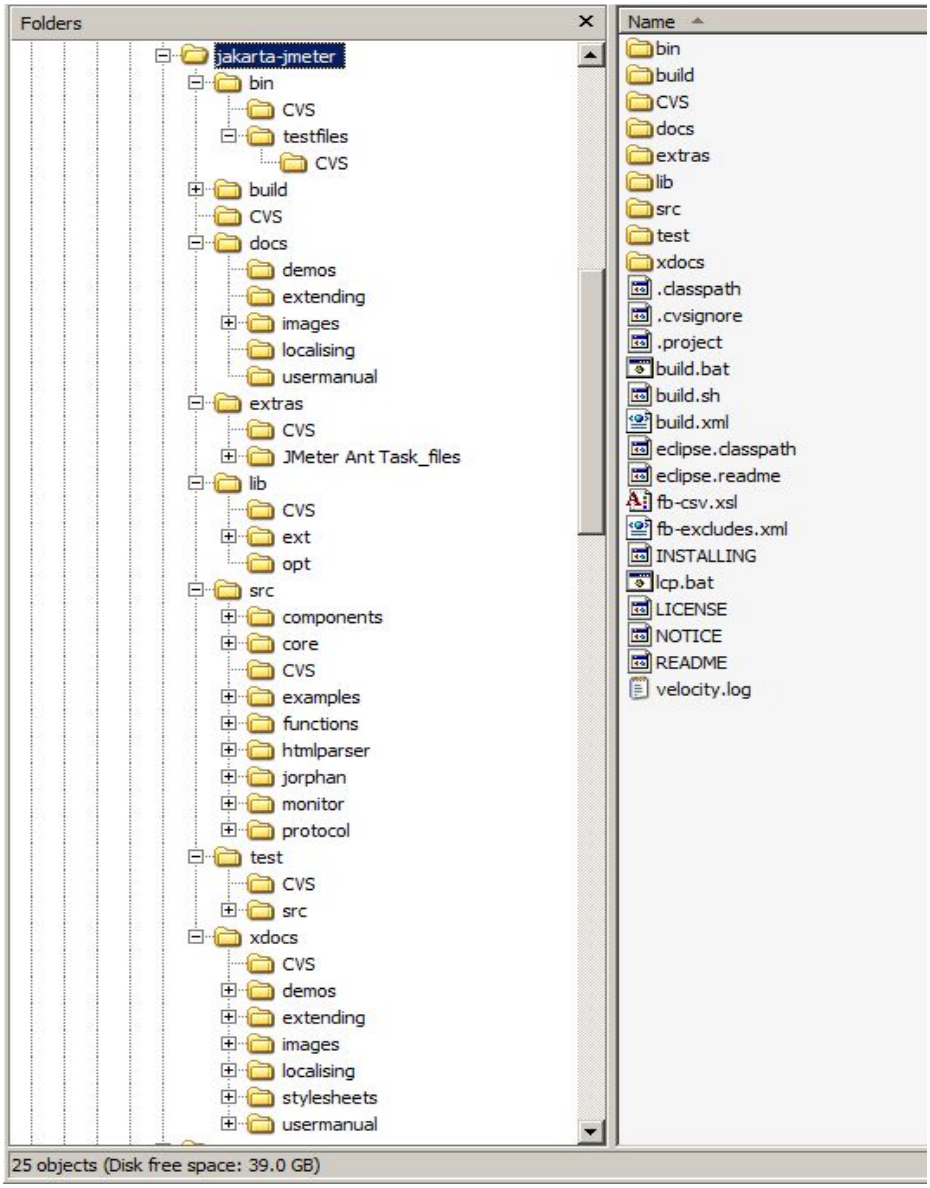
One of my primary goals in designing JMeter was to make it easy to write plugins to enhance as many of JMeter's features as possible. Part of the benefit of being open-source is that a lot of people could potentially lend their efforts to improve the application. I made a conscious decision to sacrifice some simplicity in the code to make plugin writing a way of life for a JMeter developer.

While some folks have successfully dug straight into the code and made improvements to JMeter, a real tutorial on how to do this has been lacking. I tried a long time ago to write some documentation about it, but most people did not find it useful. Hopefully, with Peter's help, this attempt will be more successful.

Basic structure of JMeter

JMeter is organized by protocols and functionality. This is done so that developers can build new jars for a single protocol without having to build the entire application. We'll go into the details of building JMeter later in the tutorial. Since most of the jmeter developers use eclipse, the article will use eclipse directory as a reference point.

Root directory - /eclipse/workspace/jakarta-jmeter/



jakarta- jmeter

- ***bin – contains the .bat and .sh files for starting Jmeter. It also contains ApacheJMeter.jar and properties file***
- ***docs – directory contains the JMeter documentation files***
- ***extras – ant related extra files***
- ***lib – contains the required jar files for Jmeter***
- ***lib/ext – contains the core jar files for jmeter and the protocols***
- ***src – contains subdirectory for each protocol and component***
- ***test – unit test related directory***

- *xdocs – Xml files for documentation. JMeter generates documentation from Xml.*

As the tutorial progresses, an explanation of the subdirectories will be provided. For now, lets focus on "src" directory. From the screen capture, we see the following directories.

Src

- *components – contains non- protocol- specific components like visualizers, assertions, etc..*
- *core – the core code of JMeter including all core interfaces and abstract classes.*
- *examples – example sampler demonstrating how to use the new bean framework*
- *functions – standard functions used by all components*
- *htmlparser – a snapshot of HtmlParser, donated by HtmlParser project on sourceforge*
- *jorphan – utility classes providing common utility functions*
- *monitor – tomcat 5 monitor components*
- *protocol – contains the different protocols JMeter supports*

Within "protocol" directory, are the protocol specific components.

Protocol

- *ftp – components for load testing ftp servers*
- *http – components for load testing web servers*
- *java – components for load testing java components*
- *jdbc – components for load testing database servers using jdbc*
- *jndi – components for load testing jndi*
- *ldap – components for load testing LDAP servers*
- *mail – components for load testing mail servers*
- *tcp – components for load testing TCP services*

As a general rule, all samplers related to HTTP will reside in "http" directory. The exception to the rule is the Tomcat5 monitor. It is separate, because the

functionality of the monitor is slightly different than stress or functional testing. It may eventually be reorganized, but for now it is in its own directory. In terms of difficulty, writing visualizers is probably one of the harder plugins to write.

Jmeter Gui – TestElement Contract

When writing any Jmeter component, there are certain contracts you must be aware of – ways a Jmeter component is expected to behave if it will run properly in the Jmeter environment. This section describes the contract that the GUI part of your component must fulfill.

GUI code in Jmeter is strictly separated from Test Element code. Therefore, when you write a component, there will be a class for the Test Element, and another for the GUI presentation. The GUI presentation class is stateless in the sense that it should never hang onto a reference to the Test Element (there are exceptions to this though).

A gui element should extend the appropriate abstract class provided:

```
AbstractSamplerGui
AbstractAssertionGui
AbstractConfigGui
AbstractControllerGui
AbstractPostProcessorGui
AbstractPreProcessorGui
AbstractVisualizer
AbstractTimerGui
```

These abstract classes provide so much plumbing work for you that not extending them, and instead implementing the interfaces directly is hardly an option. If you have some burning need to not extend these classes, then you can join me in IRC where I can convince you otherwise :-).

So, you've extended the appropriate gui class, what's left to do? Follow these steps:

1. Implement `getResourceLabel()`
    1. This method should return the name of the resource that represents the title/name

of the component. The resource will have to be entered into Jmeters messages.properties file (and possibly translations as well).

2. Create your gui. Whatever style you like, layout your gui. Your class ultimately extends Jpanel, so your layout must be in your class's own ContentPane. Do not hook up gui elements to your TestElement class via actions and events. Let swing's internal model hang onto all the data as much as you possibly can.
    1. Some standard gui stuff should be added to all Jmeter gui components:
        1. call `setBorder(makeBorder())` for your class. This will give it the standard Jmeter border
        2. add the title pane via `makeTitlePanel()`. Usually this is the first thing added to your gui, and should be done in a Box vertical layout scheme, or with Jmeter's VerticalLayout class. Here is an example init() method:

```
private void init()
{
  setLayout(new BorderLayout());
  setBorder(makeBorder());

  Box box = Box.createVerticalBox();
  box.add(makeTitlePanel());
  box.add(makeSourcePanel());
  add(box,BorderLayout.NORTH);
  add(makeParameterPanel(),BorderLayout.CENTER);
}
```

3. Implement `public void configure(TestElement el)`
    1. Be sure to call `super.configure(e)`. This will populate some of the data for you, like the name of the element.
    2. Use this method to set data into your gui elements. Example:

```
public void configure(TestElement el)
{
    super.configure(el);
    useHeaders.setSelected(el.getPropertyAsBoolean
           (RegexExtractor.USEHEADERS));
    useBody.setSelected(!el.getPropertyAsBoolean
           (RegexExtractor.USEHEADERS));
```

```
    regexField.setText(el.getPropertyAsString
           (RegexExtractor.REGEX));
    templateField.setText(el.getPropertyAsString
           (RegexExtractor.TEMPLATE));
    defaultField.setText(el.getPropertyAsString
           (RegexExtractor.DEFAULT));
    matchNumberField.setText(el.getPropertyAsString
           (RegexExtractor.MATCH_NUM));
    refNameField.setText(el.getPropertyAsString
           (RegexExtractor.REFNAME));
           }
```

4. implement `public void modifyTestElement(TestElement e)`. This is where you move the data from your gui elements to the TestElement. It is the logical reverse of the previous method.
   1. Call `super.configureTestElement(e)`. This will take care of some default data for you.
   2. Example:

```
public void modifyTestElement(TestElement e)
{
    super.configureTestElement(e);
    e.setProperty(new BooleanProperty
           (RegexExtractor.USEHEADERS,useHeaders.isSelected()));
    e.setProperty
           (RegexExtractor.MATCH_NUMBER,matchNumberField.getText
           ());
    if(e instanceof RegexExtractor)
    {
        RegexExtractor regex = (RegexExtractor)e;
        regex.setRefName(refNameField.getText());
        regex.setRegex(regexField.getText());
        regex.setTemplate(templateField.getText());
        regex.setDefaultValue(defaultField.getText());
    }
}
```

5. implement `public TestElement createTestElement()`. This method should create a new instance of your TestElement class, and then pass it to the `modifyTestElement(TestElement)` method you made above.

```
public TestElement createTestElement()
    {
        RegexExtractor extractor = new RegexExtractor();
        modifyTestElement(extractor);
        return extractor;
    }
```

The reason you cannot hold onto a reference for your Test Element is because Jmeter reuses instance of gui class objects for multiple Test Elements. This saves a lot of memory. It also makes it incredibly easy to write the gui part of your new component. You still have to struggle with the layout in Swing, but you don't have to worry about creating the right events and actions for getting the data from the gui elements into the TestElement where it can do some good. Jmeter knows when to call your configure, and modifyTestElement methods where you can do it in a very straightforward way.

Writing Visualizers is somewhat of a special case, however.

Writing a Visualizer

Of the component types, visualizers require greater depth in Swing than something like controllers, functions or samplers. You can find the full source for the distribution graph in "components/org/apache/jmeter/visualizers/". The distribution graph visualizer is divided into two classes.

- ***DistributionGraphVisualizer – visualizer which Jmeter instantiates***
- ***DistributionGraph – Jcomponent which draws the actual graph***

The easiest way to write a visualizer is to do the following:

1. ***extend org.apache.jmeter.visualizers.gui.AbstractVisualizer***
2. ***implement any additional interfaces need for call back and event notification. For example, the DistributionGraphVisualizer implements the following interfaces:***
    1. ***ImageVisualizer***

2. *ItemListener – according to the comments in the class, ItemListener is out of date and isn't used anymore.*
3. *GraphListener*
4. *Clearable*

AbstractVisualizer provides some common functionality, which most visualizers like "graph results" use. The common functionality provided by the abstract class includes:

- *configure test elements – This means it create a new ResultCollector, sets the file and sets the error log*
- *create the stock menu*
- *update the test element when changes are made*
- *create a file panel for the log file*
- *create the title panel*

In some cases, you may not want to display the menu for the file textbox. In that case, you can override the "init()" method. Here is the implementation for DistributionGraphVisualizer.

```
1    /**
2     * Initialize the GUI.
3     */
4    private void init()
5    {
6        this.setLayout(new BorderLayout());
7
8        //  MAIN PANEL
9        Border margin = new EmptyBorder(10, 10, 5, 10);
10
11       this.setBorder(margin);
12
```

```
13      // Set up the graph with header, footer, Y axis and graph display
14      JPanel graphPanel = new JPanel(new BorderLayout());
15      graphPanel.add(createGraphPanel(), BorderLayout.CENTER);
16      graphPanel.add(createGraphInfoPanel(), BorderLayout.SOUTH);
17
18      // Add the main panel and the graph
19      this.add(makeTitlePanel(), BorderLayout.NORTH);
20      this.add(graphPanel, BorderLayout.CENTER);
21  }
```

The first thing the init method does is create a new BorderLayout. Depending on how you want to layout the widgets, you may want to use a different layout manager. Keep mind using different layout managers is for experts.

The second thing the init method does is create a border. If you want to increase or decrease the border, change the four integer values. Each integer value represents pixels. If you want your visualizer to have no border, skip lines 9 and 11. Line 15 calls "createGraphPanel," which is responsible for configuring and adding the DistributionGraph to the visualizer.

```
1   private Component createGraphPanel()
2   {
3       graphPanel = new JPanel();
4       graphPanel.setBorder(BorderFactory.createBevelBorder(
5        BevelBorder.LOWERED,Color.lightGray,Color.darkGray));
6       graphPanel.add(graph);
7       graphPanel.setBackground(Color.white);
8       return graphPanel;
9   }
```

At line 6, the graph component is added to the graph panel. The constructor is where a new instance of DistributionGraph is created.

```
public DistributionGraphVisualizer()
{
    model = new SamplingStatCalculator("Distribution");
    graph = new DistributionGraph(model);
    graph.setBackground(Color.white);
                init();
}
```

The constructor of DistributionGraphVisualizer is responsible for creating the model and the graph. Every time a new result is complete, the engine passes the result to all the listeners by calling add(SampleResult res). The visualizer passes the new SampleResult to the model.

```
1   public synchronized void add(SampleResult res)
2   {
3       model.addSample(res);
4       updateGui(model.getCurrentSample());
5   }
```

In the case of the DistributionGraphVisualizer, the "add" method doesn't acutally update the graph. Instead, it calls "updateGui" in line four.

```
public synchronized void updateGui(Sample s)
{
    //  We have received one more sample
    if (delay ==  counter){
       updateGui();
       counter = 0;
    } else {
       counter++;
    }
```

```
}
```

Unlike GraphVisualizer, the distribution graph attempts to show how the results clump; therefore the DistributionGraphVisualizer delays the update. The default delay is 10 sampleresults.

```
1   public synchronized void updateGui()
2   {
3       if (graph.getWidth() < 10){
4           graph.setPreferredSize(new Dimension(getWidth() - 40, getHeight() -
160));
5       }
6       graphPanel.updateUI();
7       graph.repaint();
8   }
```

Lines 3 to 4 are suppose to resize the graph, if the user resizes the window or drags the divider. Line 6 updates the panel containing the graph. Line 7 triggers the update of the DistributionGraph. Before we cover writing graphcs, there are couple of important methods visualizer must implement.

```
public String getLabelResource()
{
   return "distribution_graph_title";
}
```

The label resource retrieves the name of the visualizer from the properties file. The file is located in “core/org/apache/jmeter/resources”. It's best not to hardcode the name of the visualizer. Message.properties file is organized alphabetically, so adding a new entry is easy.

```
public synchronized void clear()
```

```
{
    this.graph.clear();
    model.clear();
    repaint();
}
```

Every component in Jmeter should implement logic for “clear()” method. If this isn't done, the component will not clear the UI or model when the user tries to clear the last results and run a new test. If clear is not implemented, it can result in a memory leak.

```
    public JComponent getPrintableComponent(){
            return this.graphPanel;
    }
```

The last method visualizers should implement is “getPrintableComponent()”. The method is responsible for returning the Jcomponent that can be saved or printed. This feature was recently added so that users can save a screen capture of any given visualizer.

GraphListener

Visualizers should implement GraphListener. This is done to make it simpler to add new Sample instances to listeners. As a general rule, if the a custom graph does not plot every single sample, it does not need to implement the interface.

```
public interface GraphListener
{
   public void updateGui(Sample s);
   public void updateGui();
}
```

The important method in the interface is "updateGui(Sample s)". From DistributionGraphVisualizer, we see it calls graph.repaint() to refresh the graph. In most cases, the implementation of updateGui(Sample s) should do just that.

ItemListener

Visualizers generally do not need to implement this interface. The interface is used with combo boxes, checkbox and lists. If your visualizer uses one of these and needs to know when it has been updated, the visualizer will need to implement the interface. For an example of how to implement the interface, please look at GraphVisualizer.

Writing Custom Graphs

For those new to Swing and haven't written custom Jcomponents yet, I would suggest getting a book on Swing and get a good feel for how Swing widgets work. This tutorial will not attempt to explain basic Swing concepts and assumes the reader is already familiar with the Swing API and MVC (Model View Controller) design pattern. From the constructor of DistributionGraphVisualizer, we see a new instance of DistributionGraph is created with an instance of the model.

```
public DistributionGraph(SamplingStatCalculator model)
{
    this();
    setModel(model);
}
```

The implementation of "setModel" method is straight forward.

```
private void setModel(Object model)
{
    this.model = (SamplingStatCalculator) model;
    repaint();
}
```

Notice the method calls “repaint” after it sets the model. If “repaint” isn't called, it can cause the GUI to not draw the graph. Once the test starts, the graph would redraw, so calling “repaint” isn't critical.

```
public void paintComponent(Graphics g)
{
    super.paintComponent(g);
    final SamplingStatCalculator m = this.model;
    synchronized (m)
    {
      drawSample(m, g);
    }
}
```

The other important aspect of updating the widget is placing the call to drawSample within a synchronized block. If drawSample wasn't synchronized, Jmeter would throw a ConcurrentModificationException at runtime. Depending on the test plan, there may be a dozen or more threads adding results to the model. The synchronized block does not affect the accuracy of each individual request and time measurement, but it does affect Jmeter's ability to generate large loads. As the number of threads in a test plan increases, the likelihood a thread will have to wait until the graph is done redrawing before starting a new request increases. Here is the implementation of drawSample.

```
private void drawSample(SamplingStatCalculator model, Graphics g)
{
```

```
width = getWidth();
double height = (double)getHeight() - 1.0;

// first lets draw the grid
for (int y=0; y < 4; y++){
    int q1 = (int)(height - (height * 0.25 * y));
    g.setColor(Color.lightGray);
    g.drawLine(xborder,q1,width,q1);
    g.setColor(Color.black);
    g.drawString(String.valueOf((25 * y) + "%"),0,q1);
}
g.setColor(Color.black);
// draw the X axis
g.drawLine(xborder,(int)height,width,(int)height);
// draw the Y axis
g.drawLine(xborder,0,xborder,(int)height);
// the test plan has to have more than 200 samples
// for it to generate half way decent distribution
// graph. the larger the sample, the better the
// results.
if (model != null && model.getCount() > 50){
    // now draw the bar chart
    Number ninety = model.getPercentPoint(0.90);
    Number fifty = model.getPercentPoint(0.50);

    total = model.getCount();
    Collection values = model.getDistribution().values();
    Object[] objval = new Object[values.size()];
    objval = values.toArray(objval);
    // we sort the objects
    Arrays.sort(objval,new NumberComparator());
    int len = objval.length;
```

```
for (int count=0; count < len; count++){
   //  calculate the height
   Number[] num = (Number[])objval[count];
   double iper = (double)num[1].intValue()/(double)total;
   double iheight = height * iper;
   //  if the height is less than one, we set it
   //  to one pixel
   if (iheight < 1){
      iheight = 1.0;
   }
   int ix = (count * 4) + xborder + 5;
   int dheight = (int)(height -  iheight);
   g.setColor(Color.blue);
   g.drawLine(ix - 1,(int)height,ix - 1,dheight);
   g.drawLine(ix,(int)height,ix,dheight);
   g.setColor(Color.black);
   //  draw a red line for 90% point
   if (num[0].longValue() ==  ninety.longValue()){
      g.setColor(Color.red);
      g.drawLine(ix,(int)height,ix,55);
      g.drawLine(ix,(int)35,ix,0);
      g.drawString("90%",ix -  30,20);
      g.drawString(String.valueOf(num[0].longValue()),ix + 8, 20);
   }
   //  draw an orange line for 50% point
   if (num[0].longValue() ==  fifty.longValue()){
      g.setColor(Color.orange);
      g.drawLine(ix,(int)height,ix,30);
      g.drawString("50%",ix -  30,50);
      g.drawString(String.valueOf(num[0].longValue()),ix + 8, 50);
   }
}
```

```
    }
  }
```

In general, the rendering of the graph should be fairly quick and shouldn't be a bottleneck. As a general rule, it is a good idea to profile custom plugins. The only way to make sure a visualizer isn't a bottleneck is to run it with a tool like Borland OptimizeIt. A good way to test a plugin is to create a simple test plan and run it. The heap and garbage collection behavior should be regular and predictable.

Making a TestBean Plugin For Jmeter

In this part, we will go through the process of creating a simple component for Jmeter that uses the new **TestBean** framework.
This component will be a CSV file reading element that will let users easily vary their input data using csv files. To most effectively use this tutorial, open the three files specified below (found in Jmeter's **`src/components`** directory).

1. Pick a package and make three files:
   - [ComponentName].java (org.apache.jmeter.config.CSVDataSet.java)
   - [ComponentName]BeanInfo.java (org.apache.jmeter.config.CSVDataSetBeanInfo.java)
   - [ComponentName]Resources.properties (org.apache.jmeter.config.CSVDataSetResources.properties)

2) CSVDataSet.java must implement the TestBean interface. In addition, it will extend ConfigTestElement, and implement LoopIterationListener.
   - TestBean is a marker interface, so there are no methods to implement.
   - Extending ConfigTestElement will make our component a Config element in a test plan. By extending different abstract classes, you can control the type of element your component will be (ie AbstractSampler, AbstractVisualizer, GenericController, etc - though you can also make different types of elements just by instantiating the right interfaces, the abstract classes can make your life easier).

3) CSVDataSetBeanInfo.java should extend org.apache.jmeter.testbeans.BeanInfoSupport
    - create a zero-parameter constructor in which we call super(CSVDataSet.class);
    - we'll come back to this.

4) CSVDataSetResources.properties - blank for now

5) Implement your special logic for you plugin class.
    a) The CSVDataSet will read a single CSV file and will store the values it finds into JMeter's running context. The user will define the file, define the variable names for each "column". The CSVDataSet will open the file when the test starts, and close it when the test ends (thus we implement TestListener). The CSVDataSet will update the contents of the variables for every test thread, and for each iteration through its parent controller, by reading new lines in the file. When we reach the end of the file, we'll start again at the beginning. When implementing a TestBean, pay careful attention to your properties. These properties will become the basis of a gui form by which users will configure the CSVDataSet element.
    b) Your element will be cloned by JMeter when the test starts. Each thread will get it's own instance. However, you will have a chance to control how the cloning is done, if you need it.

    c) Properties: filename, variableNames. With public getters and setters.
        - filename is self-explanatory, it will hold the name of the CSV file we'll read
        - variableNames is a String which will allow a user to enter the names of the variables we'll assign values to. Why a String? Why not a Collection? Surely users will need to enter multiple (and unknown number)variable names? True, but if we used a List or Collection, we'd have to write a gui component to handle collections, and I just want to do this quickly. Instead, we'll let users input comma-delimited list of variable names.

    d) I then implemented the IterationStart method of the LoopIterationListener interface. The point of this "event" is that your component is notified of when the test has entered it's parent controller. For our purposes, every time the CSVDataSet's parent controller is entered, we will read a new line of the data file and set the variables. Thus, for a regular controller, each loop through the

test will result in a new set of values being read. For a loop controller, each iteration will do likewise. Every test thread will get different values as well.

6) Setting up your gui elements in CSVDataSetBeanInfo:

- You can create groupings for your component's properties. Each grouping you create needs a label and a list of property names to include in that grouping. Ie:

```
a)    createPropertyGroup("csv_data",new String[]{"filename","variableNames"});
```

- Creates a grouping called "csv_data" that will include gui input elements for the "filename" and "variableNames" properties of CSVDataSet. Then, we need to define what kind of properties we want these to be:

```
p = property("filename");
p.setValue(NOT_UNDEFINED, Boolean.TRUE);
p.setValue(DEFAULT, "");
p.setValue(NOT_EXPRESSION,Boolean.TRUE);
p = property("variableNames");
p.setValue(NOT_UNDEFINED, Boolean.TRUE);
p.setValue(DEFAULT, "");
p.setValue(NOT_EXPRESSION,Boolean.TRUE);
```

This essentially creates two properties whose value is not allowed to be null, and whose default values are "". There are several such attributes that can be set for each property. Here is a rundown:

```
NOT_UNDEFINED  : The property will not be left null.
DEFAULT        : A default values must be given if NOT_UNDEFINED
                 is true.
NOT_EXPRESSION : The value will not be parsed for functions if
                 this is true.
NOT_OTHER      : This is not a free form entry field - a list of
                 values has to be provided.
```

```
TAGS            : With a String[] as the value, this sets up a
                  predefined list of acceptable values, and
                  JMeter will create a dropdown select.
```

Additionally, a custom property editor can be specified for a property:

```
p.setPropertyEditorClass(FileEditor.class);
```

This will create a text input plus browse button that opens a dialog for finding a file. Usually, complex property settings are not needed, as now. For a more complex example, look at org.apache.jmeter.protocol.http.sampler.AccessLogSamplerBeanInfo

7) Defining your resource strings. In CSVDataSetResources.properties we have to define all our string resources. To provide translations, one would create additional files such as CSVDataSetResources_ja.properties, and CSVDataSetResources_de.properties. For our component, we must define the following resources:

```
displayName - This will provide a name for the element that will
              appear in menus.
csv_data.displayName - we create a property grouping called
           "csv_data", so we have to provide a label for the
           grouping
filename.displayName - a label for the filename input element.
filename.shortDescription - a tool-tip-like help text blurb.
variableNames.displayName - a label for the variable name input
           element.
variableNames.shortDescription - tool tip for the variableNames
           input element.
```

8) Debug your component.

Building JMeter

Like other Jakarta projects, Jmeter uses ANT to compile and build the distribution. Jmeter has several tasks defined, which make it easier for developers. For those unfamiliar with ANT, it's a build tool similar to make on Unix. A list of the ANT tasks with a short description is provided below.

all – builds all components and protocols
compile – compiles all the directories and components
compile- core – compiles the core directory and all dependencies
compile- components – compiles the components directory and all dependencies
compile- ftp – compiles the samples in ftp directory and all dependencies
compile- functions – compiles the functions and all dependencies
compile- htmlparser – compiles htmlparser and all dependencies
compile- http – compiles the samplers in http directory and all dependencies
compile- java – compiles the samplers in java directory and all dependencies
compile- jdbc – compiles the samplers in jdbc directory and all dependencies
compile- jorphan – compiles the jorphan utility classes
compile- ldap – compiles the samplers in ldap directory and all dependencies
compile- monitor – compiles the sampler in monitor directory and all dependencies
compile- rmi – compiles the samplers in rmi directory and all dependencies
compile- tests – compiles the tests and all dependencies
docs- api – creates the javadocs
docs- all – generates all docs.
package – compiles everythng and creates jar files of the compiled protocols
package- only – creates jar files of the compiled components

Here are some example commands.

Command	*description*
ant compile- http	*Compiles just http components*

Command	*description*
ant package	*Creates the jar files*
ant docs- all	*Generates the html documentation and javadocs*